

# Advanced Bloom Filter Based Algorithms for Efficient Approximate Data De-Duplication in Streams

Suman K. Bera

*IBM Research, India*

Sourav Dutta

*IBM Research, India*

Ankur Narang

*IBM Research, India*

Souvik Bhattacharjee\*

*University of Maryland, College Park, USA*

---

## Abstract

Data intensive applications and computing has emerged as a central area of modern research with the explosion of data stored world-wide. Applications involving telecommunication call data records, web pages, online transactions, medical records, stock markets, climate warning systems, etc., necessitate efficient management and processing of such massively exponential amount of data from diverse sources. Duplicate detection and removal of redundancy from such multi-billion datasets helps in resource and compute efficiency for downstream processing. De-duplication or *Intelligent Compression* in streaming scenarios for approximate identification and elimination of duplicates from such unbounded data stream is a greater challenge given the real-time nature of data arrival. Stable Bloom Filters (SBF) addresses this problem to a certain extent. However, SBF suffers from a high false negative rate and slow convergence rate, thereby rendering it inefficient for applications with low false negative rate tolerances.

---

**\*This work was completed at IBM Research, India.**

Email addresses: sumanber@in.ibm.com (Suman K. Bera),  
sodutta7@in.ibm.com (Sourav Dutta), annarang@in.ibm.com (Ankur Narang),  
bsouvik@cs.umd.edu (Souvik Bhattacharjee)

In this work, we present several novel algorithms for the problem of approximate detection of duplicates in data streams. We propose the *Reservoir Sampling based Bloom Filter* (RSBF) combining the working principle of reservoir sampling and Bloom Filters. We also present variants of the novel *Biased Sampling based Bloom Filter* (BSBF) based on biased sampling concepts. Using different updation and biasing mechanisms we propose variants of the same model enabling the data structure to adapt to various input scenarios. We also propose a randomized load balanced variant of the sampling Bloom Filter approach to efficiently tackle the duplicate detection. In this work, we thus provide a generic framework for de-duplication using Bloom Filters. Using detailed theoretical analysis we prove analytical bounds on the false positive rate, false negative rate and convergence rate of the proposed structures. We exhibit that our models clearly outperform the existing methods. We also demonstrate empirical analysis of the structures using real-world datasets (3 million records) and also with synthetic datasets (1 billion records) capturing various input distributions.

**Keywords:** De-duplication, Reservoir Sampling, Bloom Filter, Biased Sampling, Data streams

---

## 1. Introduction and Motivation

Data intensive computing has emerged as a central research theme in the databases and data streams community. With the tremendous spurt in the amount of data generated across varied applications, such as information retrieval, online transaction records, telecommunication call data records (CDR), virus databases, climate warning systems, web-pages and medical records to name a few, efficiently processing and managing such huge store of data has become a necessity. The problem is further compounded by the presence of spurious duplicates or redundant informations, leading to wastage to precious store space and compute efficiency. Hence, removal of such duplicates help to improve the resource utilization and compute power especially in the context of data streams requiring real-time processing at 1 GB/sec or even higher. In this work, we propose efficient algorithms to tackle the problem of real-time elimination of duplicate records present in large streaming applications. Formally, this is referred to as the *data de-duplication* or *Intelligent Compression* problem, and we use the terms interchangeably.

A national telecommunication network generates call data records, (CDR) storing important information such as the callee number, caller number, duration,

etc., for future utility. However, redundant or duplicate records may be generated due to errors in the procedure. Storing of such billions of CDR in real-time in the central data repository calls for duplicate detection and removal to enhance performance. Typical approaches involving the use of database queries or Bloom Filter [1] are prohibitively slow or are extremely resource intensive requiring around 20 GB for storing 6 billion CDR. Even disk-based algorithms have a heavy performance impact. Hence, there is a paramount need for deduplication algorithms involving in-memory operations, real-time performance along with tolerable false positive, (FP) and false negative, (FN) rates.

The growth of search engines provide another field of application for the deduplication algorithms. The search engines need to regularly crawl the Web to extract new URLs and update their corpus. Given, a list of extracted URLs, the search engines needs to perform a probe of its corpus to identify if the current URL is already present in its corpus [2]. This calls for efficient duplicate detection, wherein a small performance hit can be tolerated. A high FNR, leading to recrawling of a URL, will lead to a severe performance degradation of the search engine and a high FPR, leading to new URLs being ignored, will produce a stale corpus. Hence a balance in both FPR and FNR needs to be targeted.

Another interesting application for approximate duplicate detection in streaming environment is the detection of fraudulent advertiser clicks [3]. In web advertising domain, for the sake of profit it is possible that the publisher fakes a certain amount of the clicks (using scripts). The advertising commission necessarily need to detection such malpractices. Detection of same user ID or click generation IP in these cases can help minimizing frauds.

Straight-forward approaches to tackle this problem involving pair-wise string comparisons leads to quadratic time complexity prohibiting real-time performance. To address this issue, Bloom Filter are typically used in such domains. However, this involves huge memory requirements for tolerable performance of the algorithms and hence led to disk-based Bloom Filter approaches which again suffers from reduced throughput due to disk access overhead.

In order to address these challenges, we present the design of novel Bloom Filter based algorithms based on biased sampling, Reservoir sampling, and load based sampling. We theoretically analyze the performance of our algorithms and prove it to outperform the competing methods. We also show exhaustive empirical results to validate the enhanced performance of our methods in real-time. Using huge datasets of the order of billions of records, we portray better FPR, FNR and convergence to stability of the algorithms.

In the next section we present the related work and existing methods in this

problem domain, and discuss the various techniques used in this work. Section 3 presents the working details of the *Reservoir Sampling based Bloom Filter* algorithm, wherein we provide a novel hybrid approach based on Reservoir sampling coupled with Bloom Filter. To the best of our knowledge this is the first such attempt at combining the two for deduplication applications. It is followed by the biased sampling based techniques, *Biased Sampling based Bloom Filter* approaches, where biased sampling functions are used to operate on the Bloom Filters. Section 5 presents a randomized load balanced approach involving the load of each Bloom Filter to model its response towards each input element. We next present detailed experimental results on both real and synthetic datasets to exhibit the efficient performance of the proposed techniques. Finally, Section 7 concludes the work and provide possible future direction of work in this area.

## 2. Background and Related Work

Duplicate detection provides a classical problem within the ambit of data storage and databases giving rise to numerous buffering solutions. The advent of online arrival of data and transactions, detection of duplicates in such streaming environment using buffering and caching mechanisms [4] corresponds to a naïve solution given the inability to store all the data arriving on the stream. This led to the design of fuzzy duplicate detection mechanisms [5, 6].

Management of large data streams for computing approximate frequency moments [7], element classification [8], correlated aggregate queries [9] and others with limited memory and acceptable error rates have become a spotlight among the research community. *Bit Shaving*, the problem of fraudulent advertisers not paying commission for a certain amount of the traffic or hits have also been studied in this context [10]. This prompted the growth of approximate duplicate detection techniques in the area of both databases and web applications. Redundancy removal algorithms for search engines were first studied in [11, 12, 13]. File-level hashing was used in storage systems to help detect duplicates [14, 15, 16], but they provided a low compression ratio. Even secure hashes were proposed for fixed-sized data blocks [17].

Bloom Filters were first used by TAPER system [18]. A Bloom Filter is a space-efficient probabilistic bit-vector data structure that is widely used for membership queries on sets [19]. Typical Bloom Filter approaches involve  $k$  comparisons for each record, where  $k$  is the number of hash functions used per record for checking the corresponding bit positions of the Bloom Filter array. However, the efficiency of Bloom Filters come at the cost of a small false positive rate, wherein

the Bloom Filter falsely reports the presence of the query element. This occurs due to hash collision of multiple elements onto a single bit position of the Bloom Filter. However, there is no false negative. The probability of false positive for a standard Bloom Filter is given by [20]:

$$FPR \approx \left(1 - e^{-kn/m}\right)^k$$

Given  $n$  and  $m$ , the optimal number of hash functions  $k = \ln 2(m/n)$ .

Counting Bloom Filters [21] were introduced to support the scenario where the contents of a set change over time, due to insertions and deletions. In this approach the bits were replaced by small counters which were updated with the insert and delete of elements. However, the support for deletion operations from the structure gave rise to false negatives, where an element was wrongly reported as absent from the set. To meet the needs of varied application scenarios, a large number of Bloom Filter variants were proposed such as the compressed Bloom Filter [22], space-code Bloom Filter [23], and spectral Bloom Filter [24] to name a few. Even window model of Bloom Filters were proposed [3] such as landmark window, jumping window, sliding window [25], etc. These models operated on a definite amount of history of objects observed in the stream to draw conclusions for processing of future elements of the stream. Parallel variants of Bloom Filters were also explored.

Bloom Filters have been applied even to network related applications such as finding heavy flows for stochastically fair blue queue management [26], packet classification [27], per-flow state management and longest prefix matching [28]. Multiple Bloom Filters in conjunction with hash tables have been studied to represent items with multiple attributes accurately and efficiently with low false positive rates [29]. *Bloomjoin* used for distributed joins have also been extended to minimize network usage for query execution based on database statistics. Bloom Filters have also been used for speeding up name-to-location resolution process [30].

An interesting Bloom Filter structure proposed recently is the *Stable Bloom Filter*, SBF [31]. It provides a stable performance guarantee on a very large stream. This constant performance is of huge importance for de-duplication applications. SBF works by continuously evicting stale information from the Bloom Filters. Although it achieves a tight upper bound on FPR, the stability of the algorithm is reached theoretically at infinite stream length. In this work we present a combination of Bloom Filter and Reservoir sampling and show that the proposed method provides lower FNR, comparable FPR, but above all converges to stability much faster as compared to SBF.

Finding the number of distinct elements in a stream was explored in [32]. The problem of synopsis maintenance [33, 34] has been studied in great detail for its extensive application in query estimation [35]. Many synopsis methods such as sampling, wavelets, histograms and sketches have been designed for approximate query answering. A comprehensive survey of stream synopsis methods can be found in [36]. An important class of synopsis construction methods is the *Reservoir sampling* [37]. This sampling method has great appeal as it generates a sample of original multi-dimensional data and can be used with various data mining applications.

In Reservoir sampling one maintains a reservoir of size  $n$  from the data stream. After the first  $n$  points have been added to the reservoir, subsequent elements are inserted into the reservoir with an *insertion probability* given by  $n/t$  for the  $t^{th}$  element of the stream. An interesting characteristic of this algorithm is that it is extremely easy to implement and that all subsets of data are equi-probable to be present in the reservoir. Each data point is also associated with a bias function representing its probability to be inserted into the reservoir. Hence, the procedure can inherently capture changing behavior of the stream with different such biasing functions.

A memory-less temporal bias functions for streams for evolving streams have been proposed in [38]. Apart from  $O(1)$  processing time per stream element, incorporating the bias results in upper bounds of reservoir sizes limiting the maximum space requirement to nearly constant in most cases even for an infinitely long data stream. In this work we present several biased sampling techniques on the Bloom Filters, and also propose a randomized load balanced biasing scheme for the de-duplication problem.

### 3. Reservoir Sampling based Bloom Filter (RSBF) Approach

In this section, we propose the design and working model of the *Reservoir Sampling based Bloom Filter* (RSBF) for de-duplication in large data streams. *RSBF* intelligently combines the concepts of reservoir sampling techniques [39] and that of Bloom Filter approach. To the best of our knowledge, such an integration has not been proposed so far in the literature.

RSBF comprises  $k$  Bloom Filters, each of size  $s$  bits and are initially set to 0. On arrival of a new element,  $e$  it is hashed to one of the  $s$  bits in each of the  $k$  Bloom Filters with the help of  $k$  different uniform random hash functions. The existence of the element is verified by checking whether these  $k$  bit positions are set. If all the  $k$  bit positions are set to 1, then RSBF reports the element to be

duplicate, else to be distinct. RSBF directly inserts the initial  $s$  elements of the stream into the structure by setting the corresponding  $k$  bit positions in the Bloom Filter. Each element  $e_i$ , for  $i > s$ , is then first probed against the Bloom Filter structure to determine the duplicate or distinct status. If  $e_i$  is reported as distinct, it is inserted in the structure with probability  $p_i = s/i$  (insert probability) where  $i$  is the current length of the stream and  $s$  is the size of each of the Bloom filter.

However, with the increase in the number of bits set in the Bloom Filters, RSBF would suffer from a high rate of *false positives* wherein a distinct element is falsely reported as duplicate. As the length of the stream increases, it can be observed that the probability of an element being a duplicate increases (since the elements are drawn from a finite universe). The reservoir sampling method implicitly helps to prevent such a scenario by increasingly rejecting elements from being inserted into the structure (as the *insert probability* decreases). Insertion of elements from a possibly infinite stream would inevitable lead to the setting of nearly all the bits of RSBF to 1, thereby incurring a high false positive rate (FPR). To alleviate this problem, whenever an element is inserted into RSBF, the algorithm also deletes  $k$  randomly uniformly chosen bit (one from each Bloom Filter) by setting it to 0. It should be observed that such deletion operation invariably leads to the presence of *false negatives*, where a duplicate element is reported as distinct.

Applications involving duplicate detection demand low tolerance for both false positive as well as false negative rates (FNR). We observed that the use of reservoir sampling helps to keep the false positive rate significantly lower. However, the repeated rejection of elements (possibly distinct) with increase in the stream length may result in an increase of the FNR, thereby degrading the performance of RSBF. In order to address this problem, we introduce a weak form of biasing on the reservoir sampling operation performed on the stream elements. When the insert probability of an element decreases beyond a specified threshold,  $p^*$  and is reported as distinct by probing its bits, the element is inserted. This novel combination of reservoir sampling with thresholding thus helps to reduce FNR to acceptable limits. This procedure also helps RSBF to dynamically adapt itself to an evolving stream.

We emphasize that along with observing a low FPR and FNR, RSBF also exhibits faster convergence to stability, as compared to that of SBF, as the setting and deletion of  $k$  bits lead to a near constant number of 1's and 0's in the structure. The pseudo-code for the working of RSBF is presented in Algorithm 1 and its structure is diagrammatically represented by Fig. 1. In the following section, we provide a detailed theoretical analysis of RSBF, and later provide empirical results

to validate our claims.

---

**Algorithm 1:**  $RSBF(S)$

---

**Require:** Threshold FPR ( $FPR_t$ ), Memory in bits ( $M$ ), and Stream ( $S$ )

**Ensure:** Detecting *duplicate* and *distinct* elements in  $S$

Compute the value of  $k$  from  $FPR_t$ .  
Construct  $k$  Bloom filters each having  $M/k$  bits of memory.  
 $iter \leftarrow 1$   
**for** each element  $e$  of  $S$  **do**  
    Hash  $e$  into  $k$  bit positions,  $H = h_1, \dots, h_k$ .  
    **if** all bit positions in  $H$  are set **then**  
         $Result \leftarrow DUPLICATE$   
    **else**  
         $Result \leftarrow DISTINCT$   
    **end if**  
    **if**  $iter \leq s$  **then**  
        Set all the bit positions in  $H$ .  
    **else**  
        **if**  $(s/iter) \leq p^*$  **then**  
            **for all** positions  $h_i$  in  $H$  **do**  
                **if**  $h_i = 0$  **then**  
                    Find a bit in  $i^{th}$  bloom filter which is set to 1, and reset to 0.  
                    Set the bit at  $h_i$  position to 1  
                **end if**  
            **end for**  
        **else**  
            With probability  $(s/iter)$  insert  $e$  by setting all the bit positions in  $H$ .  
            If  $e$  was decided to be inserted then randomly reset one bit positions  
            from each of the  $k$  Bloom filters.  
        **end if**  
    **end if**  
     $iter \leftarrow iter + 1$   
**end for**

---

### 3.1. General Framework

In this section we present a generic framework for analyzing the false positive rate (FPR) and the false negative rate (FNR) of our proposed Bloom Filter based



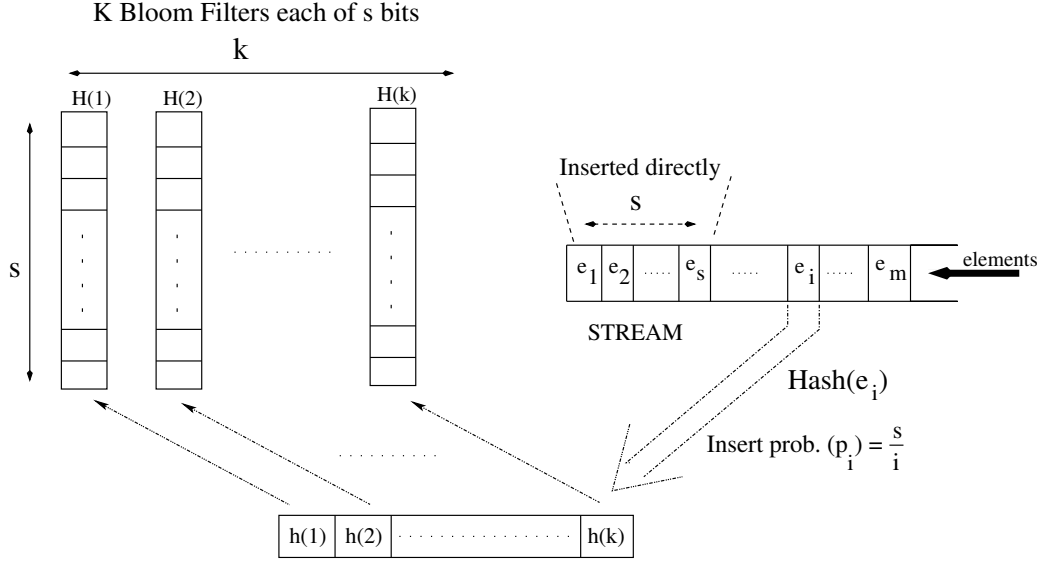


Figure 1: The working model of RSBF.

algorithms.

The event of a false positive (FP) occurs when a distinct element of the stream is reported as duplicate. A false negative (FN) event occurs when a duplicate element of the stream is reported as distinct. Now we consider the scenarios under which FP or FN can take place. Assume  $e_{m+1}$ , the  $(m+1)^{th}$  element of the stream to have arrived, and is hashed to  $H_{m+1} = h_1, h_2, \dots, h_k$  positions where  $h_i \in [1, s]$  for the  $i^{th}$  Bloom Filter.  $e_{m+1}$  will be reported as a duplicate if all the bit positions in  $H_{m+1}$  are already set to 1. Let  $X_{m+1}$  be the probability of this event. If at least one of the bit positions in  $H_{m+1}$  is 0, then  $e_{m+1}$  will be reported as distinct. Also, let us denote by  $Y_{m+1}$  the probability that  $e_{m+1}$  is actually a distinct element. Hence, we have

$$X_{m+1} = P(\text{all bit positions in } H_{m+1} \text{ are 1 when } e_{m+1} \text{ arrived}) \quad (3.1)$$

$$Y_{m+1} = P(e_{m+1} \text{ is actually a distinct element}) \quad (3.2)$$

Assume  $FPR_{m+1}$  and  $FNR_{m+1}$  denotes the probability that  $m+1^{th}$  element of the stream is a FP and FN respectively, which we show to be determined by the

quantities  $X_{m+1}$  and  $Y_{m+1}$ . So,

$$\begin{aligned}
FPR_{m+1} &= P(e_{m+1} \text{ is actually distinct}).P(e_{m+1} \text{ is reported duplicate}) \\
&= P(e_{m+1} \text{ is actually distinct}).P(\text{all bit positions in } H_{m+1} \\
&\quad \text{are 1 when } e_{m+1} \text{ arrived}) \\
&= Y_{m+1}.X_{m+1}
\end{aligned} \tag{3.3}$$

$$\begin{aligned}
FNR_{m+1} &= P(e_{m+1} \text{ is actually a duplicate}).P(e_{m+1} \text{ is reported distinct}) \\
&= P(e_{m+1} \text{ is actually a duplicate}).P(\text{not all bit positions in } \\
&\quad H_{m+1} \text{ are 1 when } e_{m+1} \text{ arrived}) \\
&= (1 - Y_{m+1}).(1 - X_{m+1})
\end{aligned} \tag{3.4}$$

For simplicity of analysis, we consider the probability of two more events: (i) the algorithm correctly predicts  $e_{m+1}$  as duplicate, (ii) the algorithm correctly predicts  $e_{m+1}$  as distinct. Let us denote by  $DUP_{m+1}$  and  $DIS_{m+1}$  the probability of the event (i) and (ii) respectively. Thus, the expression for  $DUP_{m+1}$  and  $DIS_{m+1}$  are,

$$\begin{aligned}
DUP_{m+1} &= P(e_{m+1} \text{ is actually duplicate}).P(e_{m+1} \text{ is reported duplicate}) \\
&= P(e_{m+1} \text{ is actually duplicate}).P(\text{all bit positions in } H_{m+1} \\
&\quad \text{are 1 when } e_{m+1} \text{ arrived}) \\
&= (1 - Y_{m+1}).X_{m+1}
\end{aligned} \tag{3.5}$$

$$\begin{aligned}
DIS_{m+1} &= P(e_{m+1} \text{ is actually a distinct}).P(e_{m+1} \text{ is reported distinct}) \\
&= P(e_{m+1} \text{ is actually a distinct}).P(\text{not all bit positions in } \\
&\quad H_{m+1} \text{ are 1 when } e_{m+1} \text{ arrived}) \\
&= Y_{m+1}.(1 - X_{m+1})
\end{aligned} \tag{3.6}$$

We assume that elements of the stream are uniformly randomly drawn from a finite universe  $\Gamma$ , with  $|\Gamma| = U$ . Then the probability that  $e_{m+1}$  is indeed a distinct element,  $Y_{m+1}$ , is,

$$Y_{m+1} = \left( \frac{U-1}{U} \right)^m \tag{3.7}$$

It can be observed that as  $U-1 \leq U$ ,  $Y_{m+1}$  tends to 0 as the stream length,  $m$  tends to infinity. Hence inherently the FPR for the algorithms tend to 0 with increase

in the stream length. This can be attributed to the fact that the probability of an incoming element to be distinct decreases with stream length as the elements of the stream are drawn from a finite universe. However to ensure low FNR for large streams,  $X_{m+1}$  must tend to 1 as  $m$  increases. We later show that this property is maintained by our proposed algorithms, thereby attaining a very low FNR as well as very low FPR on large data streams.

### 3.2. Analysis of RSBF

A detailed analysis of the FPR, FNR, and convergence rate of RSBF is provided in our previous work [39]. However, the impact of  $p^*$  was left as future work in that contribution. We now provide the complete analysis of RSBF including the factor  $p^*$ . Recall that for RSBF without  $p^*$ , we have

$$FPR_{m+1} = \left(\frac{U-1}{U}\right)^m \cdot \left[1 - \frac{ks}{m} + \left(\left[1 - \frac{1}{e}\right] \cdot \frac{s}{m}\right)^k\right] \quad (3.8)$$

$$FNR_{m+1} \approx O\left(\frac{k}{U}\right) \quad (3.9)$$

From equation (3.8), we observe that the right multiplicative factor tends to 1 as the stream length  $m$  reaches infinity. However, the left multiplicative factor tends to 0 as  $U-1 < U$ . Hence with increasing stream length, the FPR decreases and nearly becomes constant. From equation (3.9) we observe that the FNR becomes constant as the stream length increases. We next present the analysis for the RSBF in the presence of  $p^*$  and show that RSBF with  $p^*$  also exhibits same property. We use the generic framework discussed earlier to perform the analysis.

We now derive the expression for  $X_{m+1}$  and use it for computing  $FPR_{m+1}$  and  $FNR_{m+1}$ . Assume that  $e_{m+1}$ , the  $(m+1)^{th}$  element of the stream hashes to  $H_{m+1} = h_1, h_2, \dots, h_k$  positions where each  $h_i \in [1, s]$  for the  $i^{th}$  Bloom filter. Since all the Bloom filters are identically processed, we perform the analysis for one Bloom filter and then extends the analysis for  $k$  Bloom Filters. Assume  $l$  is the last iteration when  $h_i$  bit position of the  $i^{th}$  Bloom filter made a transition from 0 to 1 and thereafter it was never reset. The bit position  $h_i$  will not be reset after the  $l^{th}$  iteration under the following two conditions:

- (i)  $e_j(l+1 \leq j \leq m)$  is not inserted by RSBF, or,
- (ii)  $e_j(l+1 \leq j \leq m)$  is selected for insertion but some bit position other than  $h_i$  is selected for deletion from  $i^{th}$  Bloom filter.

We now define three different sequential phases in the RSBF algorithm: (i) Phase 1: During this phase all the elements of the stream is inserted into the

Bloom filter structure. This phase continues to run till  $s^th$  element of the stream is processed. (ii) Phase 2: This phase starts after Phase 1 and during this phase distinct reported elements are inserted into the Bloom filter with probability  $s/i$  where  $i$  is the current iteration. This phase ends when  $p^*$  starts operating. (iii) Phase 3: During this phase the insertion probability of an element fall below  $p^*$  and hence the distinct elements are always inserted into the Bloom Filters. We denote the position of the stream from which the effect of  $p^*$  starts taking place by  $p$ , and assume  $m + 1 > p$ .

$h_i$  is set to 1 for the last time in the  $l^{th}$  iteration and after that it is not reset. We denote the probability of this event by  $P_{trans}^l$ . Also we denote by  $NR_l^p$  the probability that  $h_i$  was not reset from iteration  $l$  to  $p$  when  $l > s$ . Let  $NR_l^m$  capture the probability that  $h_i$  is not reset in iterations from  $l$  to  $m$  when  $l > p$ .  $l$  can be in one of the three phases that we have defined above.

If  $l$  lies in the Phase 1, ( $1 \leq l \leq s$ )

$$\begin{aligned} P_{trans}^l &= P(e_l \text{ chooses } h_i).P(h_i \text{ was never reset thereafter}) \\ &= P(e_l \text{ chooses } h_i).NR_s^p.NR_p^m \end{aligned} \quad (3.10)$$

If  $l$  lies in the Phase 2, ( $s \leq l \leq p$ )

$$\begin{aligned} P_{trans}^l &= P(e_l \text{ is reported distinct}).P(e_l \text{ is inserted}).P(e_l \text{ chooses } h_i). \\ &\quad P(h_i \text{ was never reset thereafter}) \\ &= (1 - X_l) \left(\frac{s}{l}\right) \left(\frac{1}{s}\right).NR_l^p.NR_p^m \\ &= (1 - X_l) \left(\frac{1}{l}\right).NR_l^p.NR_p^m \end{aligned} \quad (3.11)$$

If  $l$  lies in the Phase 3, ( $p \leq l \leq m$ )

$$\begin{aligned} P_{trans}^l &= P(e_l \text{ is reported distinct}). \\ &\quad P(e_l \text{ chooses } h_i).P(h_i \text{ was never reset thereafter}) \\ &= (1 - X_l) \left(\frac{1}{s}\right).NR_l^m \end{aligned} \quad (3.12)$$

We now derive the expressions for  $NR_s^p$ .

$$\begin{aligned}
NR_s^p &= P(h_i \text{ was not reset after it was set before } s^{th} \text{ iteration}) \\
&= \prod_{i=s+1}^p P(e_i \text{ was reported duplicate}) + P(e_i \text{ was reported distinct}). \\
&\quad [P(e_i \text{ was not inserted}) + P(e_i \text{ was inserted}) \\
&\quad P(\text{Some other bit position than } h_i \text{ was selected for deletion})] \\
&= \prod_{i=s+1}^p \left( X_i + (1 - X_i) \left( \left( 1 - \frac{s}{i} \right) + \frac{s}{i} \cdot \frac{s-1}{s} \right) \right) \\
&= \prod_{i=s+1}^p \left( X_i + (1 - X_i) \left( 1 - \frac{1}{i} \right) \right)
\end{aligned} \tag{3.13}$$

Similarly  $NR_l^p$  where  $s \leq l < p$  is as follows, 5

$$NR_l^p = \prod_{i=l+1}^p \left( X_i + (1 - X_i) \left( 1 - \frac{1}{i} \right) \right) \tag{3.14}$$

Expressions for  $NR_p^m$  and  $NR_l^m$  are similarly derived for  $p \leq l < m$ .

$$NR_p^m = \prod_{i=p+1}^m \left( X_i + (1 - X_i) \left( 1 - \frac{1}{s} \right) \right) \tag{3.15}$$

$$NR_l^m = \prod_{i=l+1}^m \left( X_i + (1 - X_i) \left( 1 - \frac{1}{s} \right) \right) \tag{3.16}$$

It should be noted that after crossing the point  $p$ , all the distinctly reported data points are inserted and each time an insertion takes place, deletion occurs. We now derive the expression for  $X_{m+1}$ . Since the value of  $l$  can vary, we sum over all possible values of  $l$  in the different ranges. The probability that  $h_i$  was set in during the first  $s$  iterations is given by  $\{1 - (1 - \frac{1}{s})^s\}$ . Therefore we get, for  $((m+1) > p)$

$$X_{m+1} = [A_{1-s} + A_{s-p} + A_{p-m}]^k \tag{3.17}$$

where

$$A_{1-s} = \left\{ 1 - \left( 1 - \frac{1}{s} \right)^s \right\} \{NR_s^p\} \{NR_p^m\} \quad (3.18)$$

$$A_{s-p} = \sum_{l=s+1}^p \left\{ (1 - X_l) \frac{1}{l} \right\} \{NR_l^p\} \{NR_p^m\} \quad (3.19)$$

$$A_{p-m} = \sum_{l=p+1}^m \left\{ (1 - X_l) \cdot \frac{1}{s} \right\} \{NR_l^m\} \quad (3.20)$$

The probability of an FNR at point  $m+1$  where  $(m+1) < p$  follows directly from the above expression,

$$X_{m+1} = \left[ A'_{1-s} + A'_{s-m} \right]^k \quad (3.21)$$

where

$$A'_{1-s} = \left\{ 1 - \left( 1 - \frac{1}{s} \right)^s \right\} \{NR_s^{m'}\} \quad (3.22)$$

$$A'_{s-m} = \sum_{l=s+1}^p \left\{ (1 - X_l) \frac{1}{l} \right\} \{NR_l^{m'}\} \quad (3.23)$$

$$(3.24)$$

The terms  $NR_s^{m'}$  and  $NR_l^{m'}$  are obtained by carrying out similar analysis as before.

$$NR_s^{m'} = \prod_{i=s+1}^m \left( X_i + (1 - X_i) \left( 1 - \frac{1}{i} \right) \right) \quad (3.25)$$

$$NR_l^{m'} = \prod_{i=l+1}^m \left( X_i + (1 - X_i) \left( 1 - \frac{1}{i} \right) \right) \quad (3.26)$$

Careful observation of the expression for  $X_{m+1}$  provides the following recurrence relation, If  $m \leq p$ ,

$$X_{m+1} = \left[ (X_m)^{\frac{1}{k}} \left\{ X_m + (1 - X_m) \left( 1 - \frac{1}{m} \right) \right\} + (1 - X_m) \cdot \frac{1}{m} \right]^k \quad (3.27)$$

Else ( $m > p$ )

$$X_{m+1} = \left[ (X_m)^{\frac{1}{k}} \left\{ X_m + (1 - X_m) \left( 1 - \frac{1}{s} \right) \right\} + (1 - X_m) \cdot \frac{1}{s} \right]^k \quad (3.28)$$

In the following lemma we show that  $X$  is monotonically increasing and converges to 1. As  $FNR_{m+1} = (1 - Y_{m+1}) \cdot (1 - X_{m+1})$  3.1,  $RSBF$  exhibits very low FNR with increase in stream length. We later present empirical results to validate the claim that  $X$  converges to 1 along with a fast convergence rate.

**Theorem 3.1.** *For  $RSBF$   $X$  monotonically increases and converges to 1. Therefore FNR tends to 0 with increase in stream length.*

*Proof.* From Eq. (3.21) and Eq. (3.27) we observe that,  $X_1 = 0$  and  $X_2 = \frac{1}{m^k}$ . Hence, using Eq. (3.27) and Eq. (3.28) we have, For  $m \leq p$

$$\begin{aligned} \left( \frac{X_{m+1}}{X_m} \right)^{\frac{1}{k}} &= X_m + (1 - X_m) \left( 1 - \frac{1}{m} \right) + \frac{1}{X_m^{\frac{1}{k}}} \cdot (1 - X_m) \cdot \frac{1}{m} \\ &= 1 - (1 - X_m) \cdot \frac{1}{m} + (1 - X_m) \cdot \frac{1}{m \cdot X_m^{\frac{1}{k}}} \\ &= 1 + (1 - X_m) \cdot \frac{1}{m} \cdot (X_m^{-\frac{1}{k}} - 1) \end{aligned} \quad (3.29)$$

For  $m > p$

$$\begin{aligned} \left( \frac{X_{m+1}}{X_m} \right)^{\frac{1}{k}} &= X_m + (1 - X_m) \left( 1 - \frac{1}{s} \right) + \frac{1}{X_m^{\frac{1}{k}}} \cdot (1 - X_m) \cdot \frac{1}{s} \\ &= 1 - (1 - X_m) \cdot \frac{1}{s} + (1 - X_m) \cdot \frac{1}{s \cdot X_m^{\frac{1}{k}}} \\ &= 1 + (1 - X_m) \cdot \frac{1}{s} \cdot (X_m^{-\frac{1}{k}} - 1) \end{aligned} \quad (3.30)$$

We observe that the right hand side of both Eq. (3.29) and Eq. (3.30) is greater than 1 when  $X_m < 1$ . Therefore for  $X_m \leq 1$ ,  $X_{m+1} \geq X_m$  where the equality holds only if  $X_m = 1$ . Hence  $X$  monotonically increases and converges to 1. As such, FNR tends to 0 as the stream length increases.  $\square$

#### 4. Biased Sampling based Bloom Filter (BSBF) and its Variants

In this section, we propose variants of the Bloom Filter approach for de-duplication purposes. We put forth several versions of the biased sampling techniques with detailed analysis of their performance. We further show that these structures can efficiently handle evolving data streams, providing improved performance over the state-of-art. Later we exhibit empirical results to validate our claims. It can be observed that the various modifications discussed are modelled to handle various distributions of the input stream.

The *Biased Sampling based Bloom Filter* (BSBF) approach works on similar lines as that of RSBF, albeit with a small variation on the insertion criteria. BSBF inserts into its structure all the elements arriving on the stream and reported as distinct, whereas the RSBF in contrast uses the insert probability based on reservoir sampling. The insertion follows the same steps as that of RSBF, i.e.,  $k$  bits are set to 1, one from each Bloom Filter, based on  $k$  uniform hash functions. As discussed in the earlier sections, the unbounded insertion of elements leads to an increase in the FPR of the structure. To tackle this problem, BSBF on every insertion deletes  $k$  randomly uniformly selected bits, one from each of the  $k$  Bloom Filters, thereby leading to a balance between the FPR and FNR encountered. This deletion procedure is the same as that followed by RSBF. It must be noted that the chosen bit for deletion might have already been set to 0. As such, we argue in similar lines as that of RSBF for nearly a constant number of 1's and 0's in the BSBF structure. Hence, BSBF also exhibits the attractive property of faster convergence to stability, just like that of RSBF.

Since the insertion of elements in the BSBF is not restricted, with change in the nature of the input stream, BSBF also updates the bit signature of the elements stored. Hence, we can observe that BSBF implicitly captures the biased nature of the stream and dynamically adapts itself. We next present the theoretical analysis for the performance of BSBF. Algorithm 2 depicts the pseudo-code of the working



of BSBF.

---

**Algorithm 2:**  $BSBF(S)$

---

**Require:** Threshold FPR ( $FPR_t$ ), Memory in bits ( $M$ ), and Stream ( $S$ )

**Ensure:** Detecting *duplicate* and *distinct* elements in  $S$

Compute the value of  $k$  from  $FPR_t$ .

Construct  $k$  Bloom filters each having  $M/k$  bits of memory.

**for** each element  $e$  of  $S$  **do**

Hash  $e$  into  $k$  bit positions,  $H = h_1, \dots, h_k$ .

**if** all bit at positions  $H$  are set **then**

$Result \leftarrow DISTINCT$

**else**

$Result \leftarrow DUPLICATE$

**end if**

**if**  $e$  is *DISTINCT* **then**

Randomly select  $k$  bit positions  $\hat{H} = \hat{h}_1, \hat{h}_2, \dots, \hat{h}_k$  one each from the  $k$  Bloom filters.

Reset all bits in  $\hat{H}$  to 0.

Set all the bits in  $H$  to 1.

**end if**

**end for**

---

#### 4.1. Analysis of BSBF

The probability of FPR or FNR at  $e_{m+1}$  depends on the value of  $X_{m+1}$ , the probability that all bit positions in  $H_{m+1}$  are 1 when  $e_{m+1}$  arrives. Using the framework discussed earlier in section 3.1 and the values of  $FPR_{m+1}$  and  $FNR_{m+1}$ , we now derive the expression of  $X_{m+1}$  for BSBF.

The  $(m+1)^{th}$  element of the stream,  $e_{m+1}$  hashes to  $H_{m+1} = h_1, h_2, \dots, h_k$  positions, where each  $h_i \in [1, s]$  for the  $i^{th}$  Bloom filter. Initially all the bits of the Bloom filters are set to 0. Since all the Bloom filters are identical and independent, we first consider only a single Bloom filter and later extend our arguments for all the  $k$  Bloom Filters. We assume  $l$  to be the last iteration whence the  $h_i^{th}$  bit position of the  $i^{th}$  Bloom filter makes the last transition from 0 to 1, and thereafter  $h_i$  is never reset. A bit in the Bloom filter is reset by BSBF only when some element is inserted. Hence,  $h_i$  should not be reset in an iteration  $j$ , ( $l+1 \leq j \leq m$ ) if  $e_j$  is not inserted (that is  $e_j$  is reported as duplicate) or if  $e_j$  is selected for insertion and some other bit position is chosen for reset. The probability of some other bit

position to be chosen is given by  $(1 - \frac{1}{s})$ . We denote the probability of such a transition by  $P_{trans}^{(l)}$ . Hence,

$$\begin{aligned}
P_{trans}^{(l)} &= P(l \text{ is the last iteration when } h_i \text{ is set to 1 thereafter it was never reset}) \\
&= P(e_l \text{ was reported distinct}) \cdot P(e_l \text{ chooses } h_i) \cdot P(h_i \text{ was never reset thereafter}) \\
&= P(e_l \text{ was reported distinct}) \cdot P(e_l \text{ chooses } h_i) \left\{ \prod_{i=l+1}^m [P(e_i \text{ was reported duplicate}) \right. \\
&\quad \left. + P(e_i \text{ was reported distinct}) \cdot (1 - \frac{1}{s})] \right\} \\
&= \{(1 - X_l) \cdot Y_l + (1 - X_l)(1 - Y_l)\} \cdot \frac{1}{s} \cdot \left\{ \prod_{i=l+1}^m \left[ X_i + (1 - X_i)(1 - \frac{1}{s}) \right] \right\}
\end{aligned} \tag{4.1}$$

Since this transition can happen in any iteration from 1 to  $m$ ,  $l \in [1, m]$ . As the same analysis holds for all the  $k$  Bloom filters, we obtain the expression for  $X_{m+1}$  as follows,

$$\begin{aligned}
X_{m+1} &= \left[ \sum_{l=1}^m \{(1 - X_l) \cdot Y_l + (1 - X_l)(1 - Y_l)\} \cdot \frac{1}{s} \cdot \left\{ \prod_{i=l+1}^m \left[ X_i + (1 - X_i)(1 - \frac{1}{s}) \right] \right\} \right]^k \\
&= \left[ \sum_{l=1}^m (1 - X_l) \cdot \frac{1}{s} \cdot \left\{ \prod_{i=l+1}^m \left[ X_i + (1 - X_i)(1 - \frac{1}{s}) \right] \right\} \right]^k
\end{aligned} \tag{4.2}$$

Carefully observing the right hand side of the above equation, the following recurrence relation for  $X_{m+1}$  holds,

$$X_{m+1} = \left[ (X_m)^{\frac{1}{k}} \left\{ X_m + (1 - X_m) \cdot (1 - \frac{1}{s}) \right\} + (1 - X_m) \cdot \frac{1}{s} \right]^k \tag{4.3}$$

In the next lemma we show that  $X$  is monotonically increasing and converges to 1. As  $FNR_{m+1} = (1 - Y_{m+1}) \cdot (1 - X_{m+1})$  3.1, *BSBF* exhibits very low FNR with increase in stream length. We later present empirical result to validate the claim that  $X$  converges to 1 along with a fast convergence rate.

**Lemma 1.**  *$X$  monotonically increases and converges to 1. Therefore FNR tends to 0 with increase in stream length.*

*Proof.* From Eq. (4.2) we observe that,  $X_1 = 0$  and  $X_2 = \frac{1}{s^k}$ . Hence, using Eq. (4.3) we have,

$$\begin{aligned}
\left(\frac{X_{m+1}}{X_m}\right)^{\frac{1}{k}} &= X_m + (1 - X_m)\left(1 - \frac{1}{s}\right) + \frac{1}{X_m^{\frac{1}{k}}}\cdot(1 - X_m)\cdot\frac{1}{s} \\
&= 1 - (1 - X_m)\cdot\frac{1}{s} + (1 - X_m)\cdot\frac{1}{s\cdot X_m^{\frac{1}{k}}} \\
&= 1 + (1 - X_m)\cdot\frac{1}{s}\cdot(X_m^{-\frac{1}{k}} - 1)
\end{aligned} \tag{4.4}$$

We observe that the right hand side of Eq. (4.4) is greater than 1 when  $X_m < 1$ . Therefore for  $X_m \leq 1$ ,  $X_{m+1} \leq X_m$ . Hence  $X$  monotonically increases and converges to 1. As such, FNR tends to 0 as the stream length increases.  $\square$

#### 4.2. BSBF with Single Deletion

One of the major problems of deletion from a Bloom Filter arises from the fact that multiple elements may be mapped to a single bit position. Hence the deletion of that bit position (reset to 0) in practice tends to delete multiple elements from the structure. *Counting Bloom Filters* try to alleviate this problem to a certain extend, but with enormous space requirements. The heart of the problem lies in the fact that the history regarding the elements which map to a bit position is not stored in such structures. This invariably leads to a higher FNR. This provides the basic motivation for this variant of BSBF, *BSBF with Single Deletion* (BSBFSD).

The working of the BSBFSD is in close similarity with that of BSBF. The insertion procedure follows exactly the same steps as that of BSBF. However, with each insertion into the structure, we uniformly randomly select one Bloom Filter from which the bit needs to be deleted. Within this selected Bloom Filter, we again uniformly randomly choose a bit position to be set to 0. Hence BSBFSD performs deletion in a conservative manner so as to preserve as many element signature as possible. This leads to an improved performance based on FNR criteria. However, since most of the bits are set to 1 as the stream length increases, BSBFSD incurs a higher FPR compared to BSBF. However, certain application like federal crime record corpuses require extremely low or zero FNR (but can compromise with relatively higher FPR tolerance), making BSBFSD particularly suitable for such scenarios. Next we provide theoretical results claiming that the trade-off between FPR and FNR of both the BSBF and BSBFSD are better paid-off than that of SBF.

The pseudo-code for BSBFSD is provided in Algorithm 3.

---

**Algorithm 3:**  $BSBFSD(S)$

---

**Require:** Threshold FPR ( $FPR_t$ ), Memory in bits ( $M$ ), and Stream ( $S$ )

**Ensure:** Detecting *duplicate* and *distinct* elements in  $S$

Compute the value of  $k$  from  $FPR_t$ .

Construct  $k$  Bloom filters each having  $M/k$  bits of memory.

**for** each element  $e$  of  $S$  **do**

Hash  $e$  into  $k$  bit positions,  $H = h_1, \dots, h_k$ .

**if** all bit at positions  $H$  are set **then**

$Result \leftarrow DISTINCT$

**else**

$Result \leftarrow DUPLICATE$

**end if**

**if**  $e$  is *DISTINCT* **then**

Randomly select a Bloom filter  $B_i$

Randomly select a bit  $\hat{h}_i$  from the  $B_i^{th}$  Bloom Filter

Reset  $\hat{h}_i$  to 0.

Set all the bits in  $H$  to 1.

**end if**

**end for**

---

#### 4.3. Analysis of BSBFSD

It can be observed that the FPR and FNR analysis of BSBFSD is similar to that of BSBF. BSBFSD differs from BSBF in the way deletion of bits take place. In BSBF whenever a new data element is inserted into the Bloom filters, one randomly chosen bit was reset from each of the Bloom filters. In BSBFSD, after insertion we randomly select one Bloom filter and then randomly reset one bit from the selected Bloom filter. The probability that the bit  $h_i$  is not reset in an iteration involving the insertion of some element is  $(\frac{1}{k} \cdot (1 - \frac{1}{s}) + (1 - \frac{1}{k}))$  or  $(1 - \frac{1}{ks})$ . Hence the expression for  $X_{m+1}$  becomes,

$$\begin{aligned} X_{m+1} &= \left[ \sum_{l=1}^m (1 - X_l) \cdot \frac{1}{s} \cdot \left\{ \prod_{i=l+1}^m \left[ X_i + (1 - X_i) \left(1 - \frac{1}{ks}\right) \right] \right\} \right]^k \\ &= \left[ (X_m)^{\frac{1}{k}} \left\{ X_m + (1 - X_m) \cdot \left(1 - \frac{1}{ks}\right) \right\} + (1 - X_m) \cdot \frac{1}{s} \right]^k \end{aligned}$$

Similar to the analysis shown in Section 4.1,  $X$  monotonically increases to 1. The expression for  $\frac{X_{m+1}}{X_m}$  is,

$$\begin{aligned} \left(\frac{X_{m+1}}{X_m}\right)^{\frac{1}{k}} &= X_m + (1 - X_m)\left(1 - \frac{1}{ks}\right) + \frac{1}{X_m^{\frac{1}{k}}}\cdot(1 - X_m)\cdot\frac{1}{s} \\ &= 1 + (1 - X_m)\cdot\frac{1}{s}\cdot\left(X_m^{-\frac{1}{k}} - \frac{1}{k}\right) \end{aligned} \quad (4.5)$$

Using similar arguments from Lemma 1,  $X_{m+1} \geq X_m$  for  $X_m \leq 1$  (monotonically increasing) and converges to 1.

## 5. Randomized Load Balanced Biased Sampling based Bloom Filter

In this section we consider a further variation of the biased sampling based Bloom Filter approach, the *Randomized Load Balanced Biased Sampling based Bloom Filter* (RLBSBF). The main aim of this approach is to keep the load (number of 1s) in each of the Bloom Filter below a certain ratio (probabilistically) of its total space, thereby containing the FPR and FNR achieved to a low value. This would further ensure the stability of the Bloom Filter performance by keeping the count of 0 and 1 nearly constant.

The insertion procedure of an element arriving in the data stream remains the same as that of the other algorithms discussed earlier. That is, when a distinct element arrives, it is inserted into the Bloom Filters by appropriately setting the bit positions. However, there is a major difference in the deletion procedure. Whenever an element is inserted, we independently access each of the Bloom Filters and based on its current load factor probabilistically decide whether to delete (reset) a random bit or not. This approach intuitively tries to restrict FPR to a small value by limiting the number of 1s in each Bloom Filter, along with deleting as less history as possible to obtain a low FNR as well.

Formally, RLBSBF stores the load (the number of bits set) in each of the Bloom Filters to decide the probability of deletion of a bit after the insertion of an element. When an element  $e$  is reported as distinct by the algorithm, we insert  $e$  into the Bloom Filters. Like before, this is done by setting the bits is  $H$  to 1. But instead of performing deterministic deletion from the Bloom Filters, we perform randomized deletion from the Bloom Filters. From each filter we first select a random bit position and then reset it with probability  $L_{m+1}(i)/s$ , where  $L_{m+1}(i)$  denotes the load of the  $i^{th}$  Bloom filter when  $e_{m+1}$  arrived. The detailed algorithm is given in Algorithm 4.

Empirically we observed (results shown later) that this approach in turn enforces a low FPR and the lowest FNR for the competing algorithms. The efficient performance of RLBSBF can be attributed to the prevention of the number of set bits in each Bloom filter from becoming high, reducing the FPR, and on the other hand, performing load based deletion of bits (lesser deletion events) curbing down the FNR.

---

**Algorithm 4:**  $RLBSBF(S)$

---

**Require:** Threshold FPR ( $FPR_t$ ), Memory in bits ( $M$ ), and Stream ( $S$ )

**Ensure:** Detecting *duplicate* and *distinct* elements in  $S$

```

Compute the value of  $k$  from  $FPR_t$ .
Construct  $k$  Bloom filters each having  $M/k$  bits of memory.
for each element  $e$  of  $S$  do
    Hash  $e$  into  $k$  bit positions,  $H = h_1, \dots, h_k$ .
    if all bit at positions  $H$  are set then
         $Result \leftarrow DISTINCT$ 
    else
         $Result \leftarrow DUPLICATE$ 
    end if
    if  $e$  is DISTINCT then
        for all Bloom filter  $B_i$  do
            Select a random bit position  $\hat{h}_i$ .
            Reset  $\hat{h}_i$  with probability  $L(i)/s$  where  $L(i)$  is the number of ones in
            the Bloom filter  $B_i$ .
        end for
        Set all the bits in  $H$  to 1.
    end if
end for

```

---

### 5.1. Analysis of RLBSBF

For computing the probability of FNR and FPR of RLBSBF, we initially evaluate the expected load of the Bloom Filters at any iteration. Let  $L_{m+1}^i$  denotes the expected number of bits set in the  $i^{th}$  Bloom Filter when element  $e_{m+1}$  arrives and is hashed to  $h_i$  bit position of the  $i^{th}$  Bloom Filter. We are interested in finding out the probability of  $h_i$  being already set to 1 for all  $i \in [1, k]$ . This gives us  $X_{m+1}$ , which in turn determines the FPR and FNR.

We assume  $l$  to be the last iteration when  $h_i$  was set to 1 and after that it

was never reset.  $h_i$  will not be reset in an iteration  $j$  ( $l + 1 \leq j \leq m$ ) if one the following events occur: (i)  $e_j$  was reported as duplicate and hence not inserted into the Bloom Filter, (ii)  $e_j$  was inserted into the Bloom filter and with probability  $1 - \frac{L_{m+1}^i}{s}$  no bits were reset in the  $i^{th}$  Bloom Filter, (iii)  $e_j$  was inserted into the Bloom Filter and with probability  $\frac{L_{m+1}^i}{s}$  deletion from  $i^{th}$  Bloom Filter was performed, but some bit other than  $h_i$  was chosen for deletion with probability  $(1 - \frac{1}{s})$ . It can be observed that this argument applies to all the  $k$  bloom filters, and that  $l$  can vary from 1 to  $m$ . Therefore, the expression of  $X_{m+1}$  for the RLBSBF algorithm is as,

$$\begin{aligned}
X_{m+1} &= P(\text{all bit positions in } H_{m+1} \text{ are 1 when } e_{m+1} \text{ arrived}) \\
&= \left[ \sum_{l=1}^m P(l \text{ is the last iteration when } h_i \text{ is set to 1, thereafter it was never reset}) \right]^k \\
&= \left[ \sum_{l=1}^m P(e_l \text{ was reported distinct}) \cdot P(e_l \text{ chooses } h_i) \cdot P(h_i \text{ was never reset thereafter}) \right]^k \\
&= \left[ \sum_{l=1}^m (1 - X_l) \cdot \frac{1}{s} \cdot \left\{ \prod_{i=l+1}^m \left[ X_i + (1 - X_i) \left( \left(1 - \frac{L_{m+1}^i}{s}\right) + \frac{L_{m+1}^i}{s} \left(1 - \frac{1}{s}\right) \right) \right] \right\} \right]^k \\
&= \left[ \sum_{l=1}^m (1 - X_l) \cdot \frac{1}{s} \cdot \left\{ \prod_{i=l+1}^m \left[ X_i + (1 - X_i) \left(1 - \frac{L_{m+1}^i}{s^2}\right) \right] \right\} \right]^k \tag{5.1}
\end{aligned}$$

$$= \left[ (X_m)^{\frac{1}{k}} \left\{ X_m + (1 - X_m) \cdot \left(1 - \frac{L_{m+1}^i}{s^2}\right) \right\} + (1 - X_m) \cdot \frac{1}{s} \right]^k \tag{5.2}$$

We now find the expected load,  $L_{m+1}^i$  of the  $i^{th}$  Bloom Filter at the time when  $e_{m+1}$  arrived. Let us associate an indicator random variable  $I_j$  with each bit  $j$  of the Bloom filters.  $I_j$  is 1 if  $j = 1$ , otherwise 0. Also let  $Z^i$  be random variable such that  $Z^i = \sum_{j=1}^s I_j$ . Therefore  $E[Z^i]$  will give us the expected load of the  $i^{th}$

Bloom filter.

$$\begin{aligned}
L_{m+1}^i &= E[Z^i] \\
&= \sum_{j=1}^s E[I_j] \\
&= \sum_{j=1}^s P(j^{th} \text{ bit was set to 1 when } e_{m+1} \text{ arrived})
\end{aligned}$$

## 6. Experiments and Results

In this section we empirically compare the performances of SBF, RSBF, BSBF, BSBFSD and RLBSBF algorithms against various parameters like memory, stream size, percentage of distinct elements in the stream etc. We measure the performances on real dataset containing clickstream data (obtained from <http://www.sigkdd.org/kddcup/index.php?section=2000&method=data>) having around 3M elements as well as on uniformly and randomly generated datasets with upto 1B records. In all experiments  $p^*$  for RSBF has been set to 0.03. Also we present experimental results for choosing the value of  $k$  for BSBF, BSBFSD and RLBSBF algorithms. We show that our proposed algorithms are comparable or outperforms SBF with respect to FPR and FNR while exhibiting better stability and faster convergence properties.

### 6.1. Setting of Parameters

In this section we present the rationale behind setting of the parameter  $k$  (the number of Bloom filters) for the various proposed algorithms. Given a fixed memory space  $M$ , we experimentally search for an optimal value of  $k$  such that an overall low FPR and FNR is attained.

For RSBF, we have shown in our previous work [39] that

$$k = \frac{\ln(FPR_t)}{\ln(1 - \frac{1}{e})} \quad (6.1)$$

We also observed that with increase in  $k$  FPR decreases, but FNR is minimized when  $k = 1$ . As a trade-off we set  $k$  as the arithmetic mean of 1 and that obtained in (6.1). The threshold FPR  $FPR_t$  is set to 0.1.

We now present the performance of BSBF algorithm under various parametric settings of  $k$ . We chose a uniform random dataset of size 1B with 60% distinct



element. We also vary the memory size from 8MB to 512MB and analyze the FPR and FNR. From Table 1 we observe that BSBF exhibits low FNR and high FPR for  $k = 1$ . As we increase the value of  $k$ , FPR decreases while FNR increases. The increase in FNR is attributed to the fact that by increasing the value of  $k$ , more element signatures are being deleted from the Bloom Filter structures. (Every time some element is inserted, one bit from each of the  $k$  Bloom filter is reset to 0). The decrease in FPR is due to the increase in the signature length of an element in the Bloom Filter. We observe that for  $k = 2$ , both FPR and FNR attains a acceptably balanced limit. Hence for performance evaluation of BSBF algorithm, we set  $k = 2$  for the result of our experimental setup. We also observe that if higher memory space is available, then BSBF attains very low FNR (3.4%) and FPR (6.4%) for  $k = 1$ . Hence depending on the application specifications BSBF can be modeled to perform efficiently.

Algorithm:BSBF, Dataset:1B , Distinct:60%						
Space		k=1	k=2	k=3	k=4	k=5
8 MB	% FPR	75.979	35.4924	15.2961	6.95161	3.28054
	% FNR	9.20901	59.0297	82.0828	91.5512	95.7754
128 MB	% FPR	21.0883	11.9472	6.67181	3.47978	1.76989
	% FNR	9.83893	34.7514	59.2496	75.1262	84.1857
512 MB	% FPR	6.46215	1.82011	0.777613	0.386642	0.205833
	% FNR	3.34108	13.5658	27.7512	43.4681	56.9057

Table 1: Synthetic Dataset of 1B elements (60% distinct)

We next present the performance of BSBFSD algorithm across various parametric setting of  $k$ . We use the some random dataset of size 1B with 60% distinct element and vary the memory size from 8MB to 512MB. From Table 2, we observe that BSBFSD exhibits increasing FNR and decreasing FPR for increasing  $k$  when the memory size is high (128MB and 512MB). The decrease in FPR is due to increase in the signature length of an element in the Bloom Filter with increasing  $k$ . But as a trade-off FNR increases, as chances of more elements being mapped to a point of deletion increases. We observe that for  $k = 2$ , both FPR and FNR attains a reasonably balanced limit. Hence we set  $k = 2$  for BSBFSD algorithm's performance analysis.

Finally, we present the performance of RLBSBF algorithm across various parametric setting of  $k$ . We use the same random dataset of size 1B with 60%

Algorithm:BSBFSD, Dataset:1B , Distinct:60%						
Space		k=1	k=2	k=3	k=4	k=5
8 MB	% FPR	75.9827	78.6056	81.2512	83.3572	85.0294
	% FNR	9.2090	7.8180	6.4065	5.3588	4.5864
128 MB	% FPR	21.0876	15.5928	15.2615	16.4467	18.2903
	% FNR	9.8405	17.7438	22.9224	26.2752	28.3608
512 MB	% FPR	6.4618	1.9692	1.0095	0.6880	0.5590
	% FNR	3.3399	6.8855	9.9011	12.6723	15.1324

Table 2: Synthetic Dataset of 1B elements (60% distinct)

distinct element and again vary the memory size from 8MB to 512MB. From Table 3, we observe that RLBSBF exhibits similar performance as above. Likewise we set  $k = 2$ .

Algorithm:RLBSBF, Dataset:1B , Distinct:60%						
Space		k=1	k=2	k=3	k=4	k=5
8 MB	% FPR	81.4726	52.9291	30.2439	17.3562	10.1282
	% FNR	5.3103	40.466	66.0585	80.3262	88.3195
128 MB	% FPR	22.8676	16.3554	12.3957	8.56757	5.5645
	% FNR	2.4337	15.0161	35.6592	55.2835	69.3118
512 MB	% FPR	6.6563	2.07884	1.0555	0.6773	0.4828
	% FNR	0.2553	1.98971	6.1959	13.3911	23.2844

Table 3: Synthetic Dataset of 1B elements (60% distinct)

## 6.2. Quality Comparison

In this section we present the variation of FPR and FNR along with convergence to stability for SBF, RSBF, BSBF, BSBFSD and RLBSBF with increasing number of records in the input stream. We present graph based analysis of FPR and FNR performances of our algorithms in comparison with SBF for 1B data with 15% distinct element. Fig. 2 shows the graph for FPR and FNR when 128MB memory space is available. We observe that all the algorithms achieve quite low FPR in the range of 1%-2%. Also the FPR becomes stable at around 300M-350M data points for almost all the variations. However we observe a sharp contrast in

FNR level. While SBF exhibits nearly 45% FNR, RLBSBF exhibits less than 1% FNR which is almost 70x times improvement. Other variations also exhibit strong improvements in the FNR performance in comparison with SBF. We also observe that for BSBF, BSBFSD and RLBSBF the FNR level keeps on decreasing as the stream length increases. This validates our theoretical claim about low FNR for these algorithms. Fig. 3 presents the FPR and FNR for 1B random dataset with

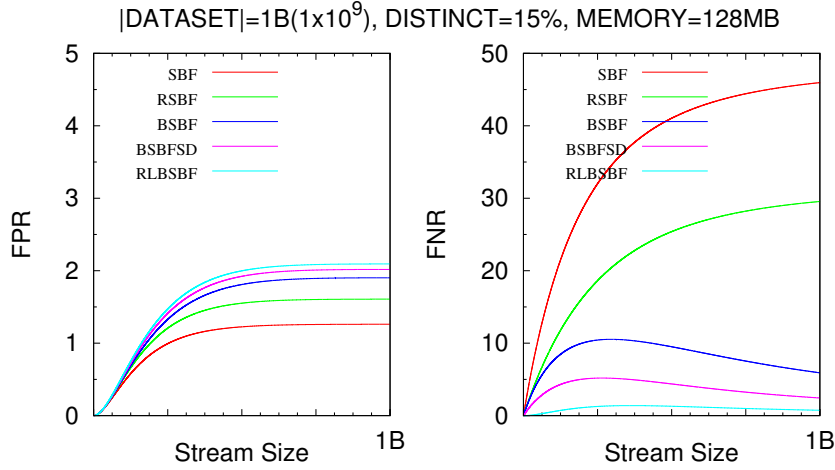


Figure 2: FPR and FNR Performances.

256MB memory space. FPR for all the algorithms falls within a small range of 0.4%-0.6%, however FNR varies from 30% for SBF to 0.2% for RLBSBF. Also the curves provide empirical evidence for our theoretical results that FNR tends to zero as we increase the stream length. SBF does not exhibit this property and its FNR increases as the stream length increases. Fig. 4 shows that as we increase memory space from 256MB to 512MB, FNR further drops to a very low limit (less than 1%) for BSBF, BSBFSD, RLBSBF. RSBF also achieves an improvement of 2x in terms of FNR over SBF. FPR of all the algorithm stabilizes at a very low level and remains comparable.

We next compare the FPR and FNR of the proposed algorithms for the 1B synthetic random dataset but with 60% distinct element. We vary the memory size from 128MB to 512MB. To emphasize the comparability of FPR and large improvements of FNR between SBF and our algorithms, we have plotted the graphs

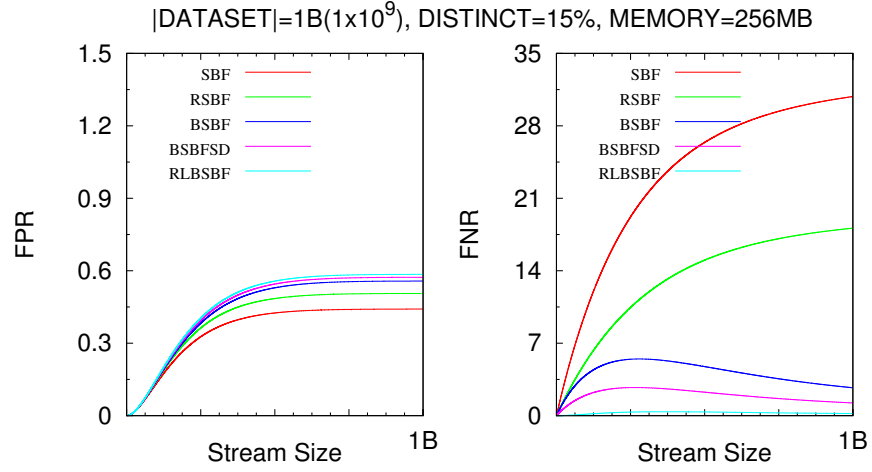


Figure 3: FPR and FNR Performances.

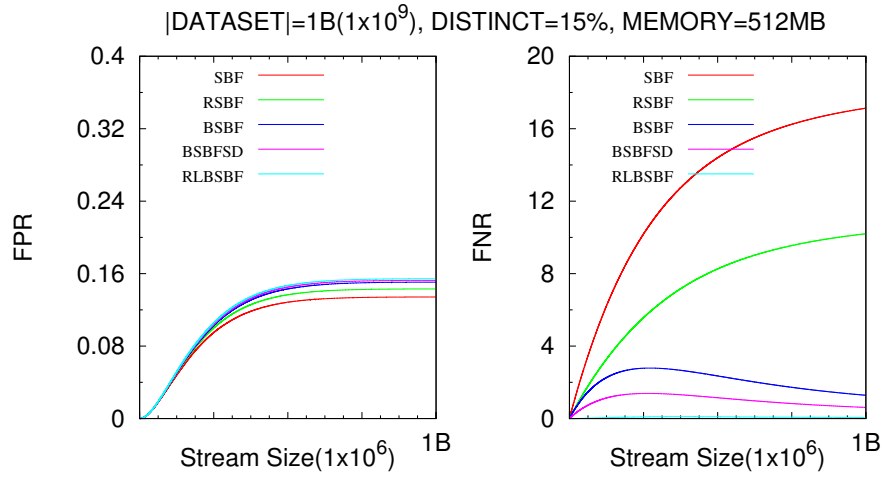


Figure 4: FPR and FNR Performances.

using natural scale. Fig. 5 shows FPR and FNR comparison for 128MB memory space. Although SBF exhibits good FPR, it has a very poor FNR(70%). RLBSBF curbs down the FNR to almost 10% while losing merely 3% on FPR. As evident from the graph, the gain over FNR dominates the loss over FPR. Also BSBF and BSBFSD attain stability in FNR ratio from around 500M data points. Fig. 6

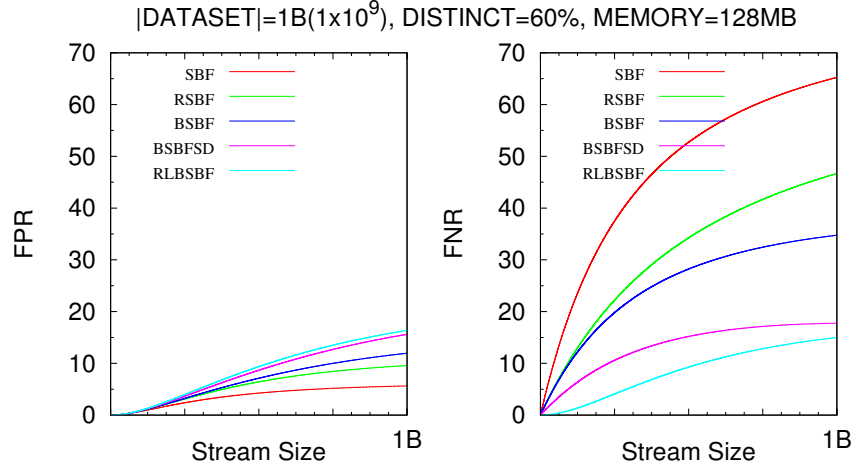


Figure 5: FPR and FNR Performances.

depicts the FPR and FNR of various algorithm for 1B data(60% distinct) when 256MB memory space is used. We observe that the gap in FPR among the algorithms have been reduced significantly. The range of FPR for the algorithms lies between 4%-7%. But the difference in FNR achieved by the algorithms is very high. RSBF, BSBF, BSBFSD and RLBSBF provides an improvement of 2x, 2.5x, 5x and 10x respectively. We also observe a similar FNR stability trend as previously for 128MB memory. Finally Fig. 7 shows that the FPR of the algorithms converges to a very low limit when 512MB memory space is used. But the difference in FNR level only keeps on improving in favor of our proposed algorithms. This also shows the scalability aspect of our algorithms. As we increase the memory space, RSBF, BSBF, BSBFSD and RLBSBF improves in performance much faster than SBF.

Figs. 8, 9 and 10 compare the FPR and FNR of the various algorithms with increasing distinct percentage in data stream keeping the memory space fixed at

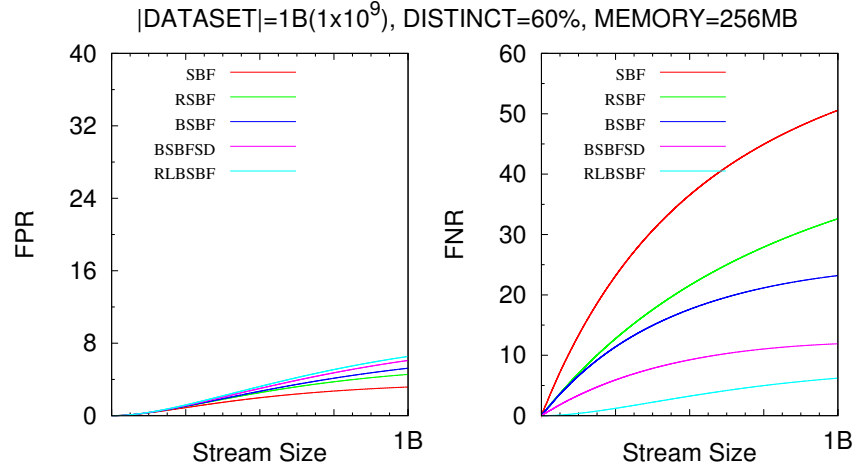


Figure 6: FPR and FNR Performances.

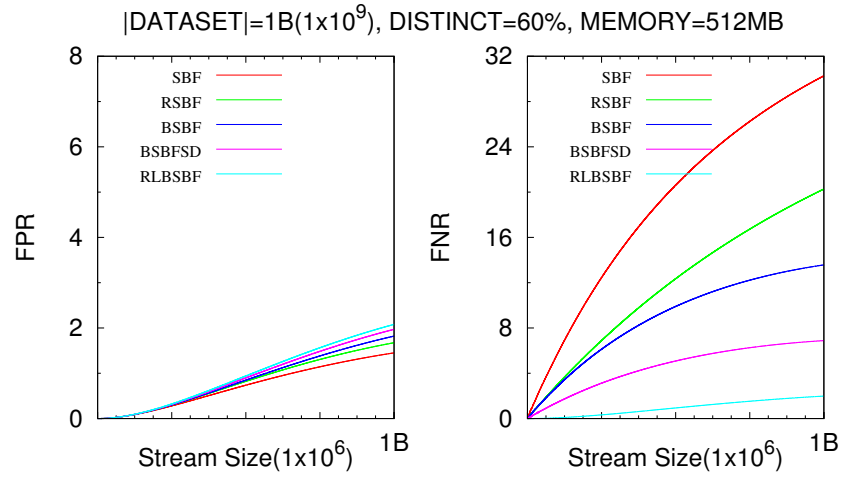


Figure 7: FPR and FNR Performances.

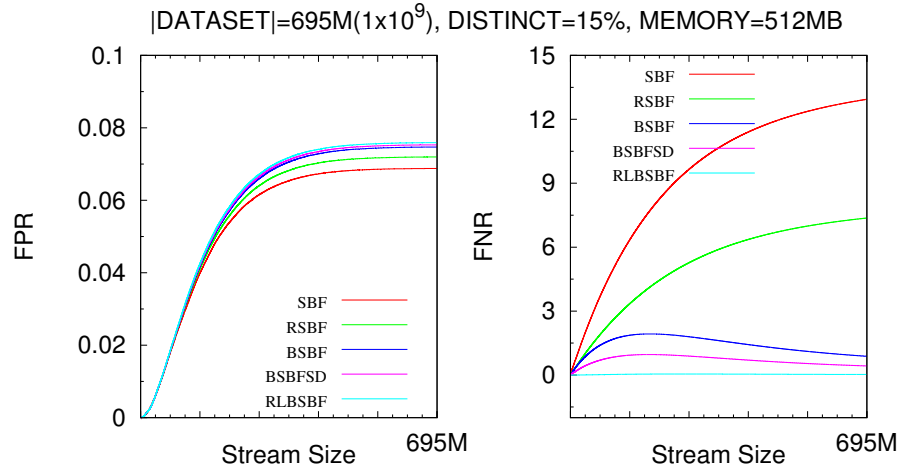


Figure 8: FPR and FNR Performances.

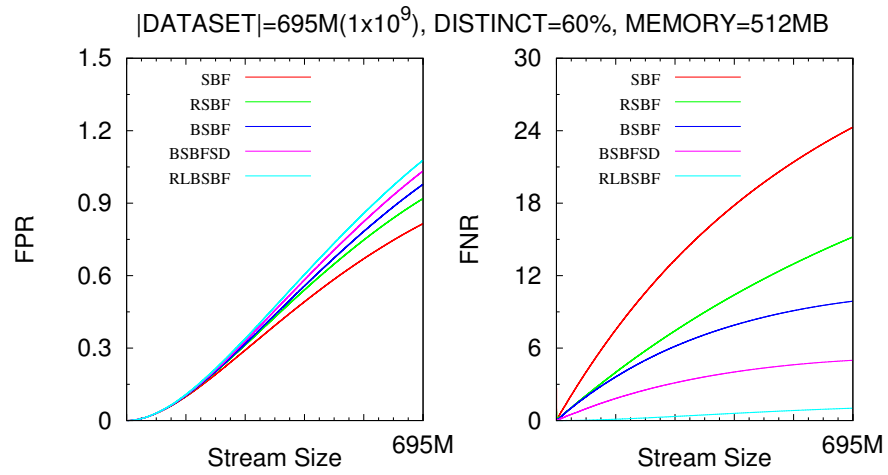


Figure 9: FPR and FNR Performances.

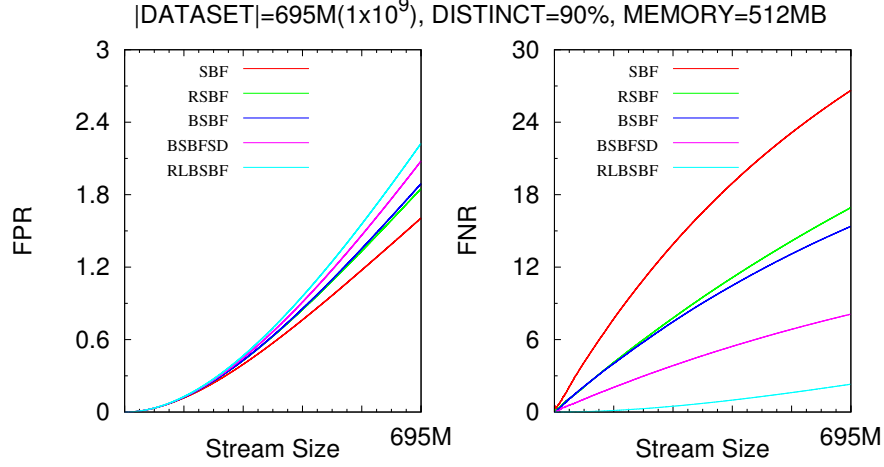


Figure 10: FPR and FNR Performances.

512MB. We generate three uniform random dataset of 695M elements with 15%, 60% and 90% distinct element. We observe that the proposed algorithms outperform SBF in FNR and convergence rates with comparable FPR.

Fig. 8 shows FPR for all the algorithm is almost similar and stable for 695M dataset with 15% distinct element. For FNR, our algorithms completely outperform SBF. Fig. 9 represents the FPR and FNR scenario for 60% distinct data stream of size 695M with 512MB size. We observe that at 695M data point, none of the algorithms have achieved stability in terms of FPR. In terms of FNR only BSBF, BSBFSD and RLBSBF are inching towards stability while SBF is far way off from stability. Fig. 10 shows the situation for 90% distinct element with similar trends as discussed above. Hence our proposed algorithms always outperform SBF comprehensively in terms of FNR and also attain comparable of FPR and stability.

We next exhibit the stability of our algorithms. We compute the load of the Bloom Filter structure at every point of the stream. We define load as the number of 1's in the Bloom Filters normalized by the total memory space in bits. We present the load graph for various algorithms for 1B data with 15% distinct. Fig. 11 shows the load graph when 256MB memory space is allocated and when 512MB memory is used. It can be observed from the graph that all the algorithms nearly attain stability when 300M-400M data points have been processed. Also,



as we increase the memory space from 256MB to 512MB, the stability is reached faster. The convergence rate of the algorithms are similar to each other.

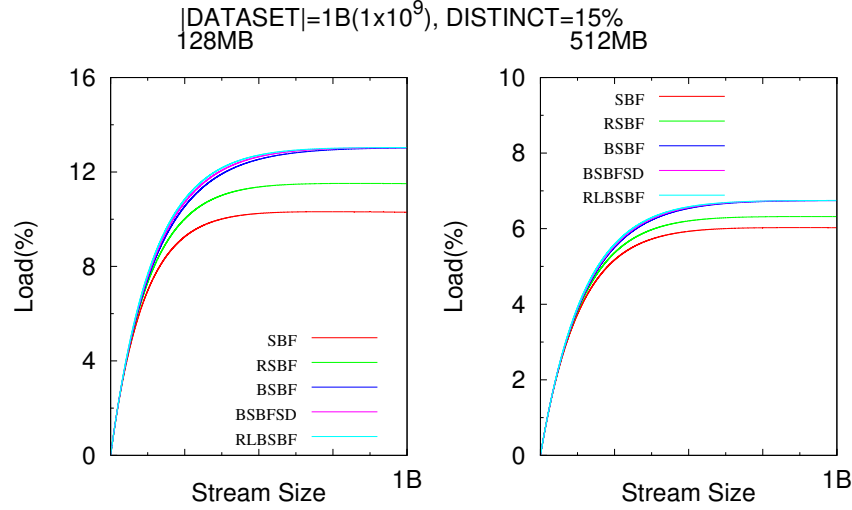


Figure 11: Stability Performance.

### 6.3. Detailed Analysis

In this section, we present detailed experimental analysis of the various algorithms that we have designed in the previous sections. We compare the performance of various algorithms against variation of memory used and percentage of distinct elements in the stream. In the following comparisons we have set  $k = 2$  for RSBF, BSBF, BSBFSD, RLBSBF algorithms.

Table 4 presents the FPR and FNR with 695M records with 15% distinct element in the stream. The memory size varies from 64MB to 512MB for the underlying Bloom Filter based data structures. Here we observe that although SBF performs reasonably well in terms of FPR for low memory (64MB), FNR degrades to an unacceptable limit (53.26%). For 64MB memory, RLBSBF keeps the FPR and FNR to very low limits (FPR of 3.7% and FNR of 1.3%). But for 128MB memory all the five algorithms exhibits comparable FPR with lowest being SBF (0.74%) and highest being RLBSBF(1.08%). But FNR for BSBF, BSBFSD and RLBSBF improve drastically compared to that of SBF. For example, RLBSBF

gains an improvement of 100x times over SBF in terms of FNR! This behavior continues for higher memory size and for 512MB memory, all the four proposed algorithms outperform SBF comprehensively in term of FNR while keeping the FPR level competitive. For example RLBSBF attains an improvement of 600x in respect to FNR when compared to SBF while FPR level for both remains almost the same.

Dataset:695M , Distinct:15%						
Space		SBF	RSBF	BSBF	BSBFSD	RLBSBF
64 MB	% FPR	1.9319	2.6276	3.2569	3.5475	3.7064
	% FNR	53.2681	35.9014	8.7547	3.3299	1.3453
128 MB	% FPR	0.7414	0.8903	1.0128	1.0530	1.0821
	% FNR	37.7853	23.126	3.888	1.7048	0.3773
256 MB	% FPR	0.2384	0.2637	0.2822	0.2881	0.2924
	% FNR	23.8930	13.5047	1.8199	0.8551	0.1010
512 MB	% FPR	0.0688	0.0719	0.0747	0.0753	0.0759
	% FNR	12.9392	7.3674	0.8794	0.4267	0.0262

Table 4: Synthetic Dataset of 695M elements (15% distinct)

Table 5 presents the FPR and FNR with 695M records with 60% distinct element in the stream. The memory size varies from 64MB to 512MB. For 8MB memory, as before SBF outperforms other algorithm in terms of FPR. But an FNR of 70.832% makes SBF unsuitable for all practical purposes. As we increase the memory size to 128MB, 256MB and 512MB, we observe sharp fall in the FPR level of all the algorithms. For 512MB, all the algorithms offers FPR that is in the range of 0.8%-1.07%. But in terms of FNR, our proposed algorithms outperforms SBF significantly. Notably, RLBSBF achieves an improvement of around 24 times in terms of FNR compared to SBF. We also observe that with increase in memory space, FNR of RLBSBF falls sharply to a very low limit. For example, if we double the memory space from 128MB to 256MB and then 256MB to 512MB, FNR of RLBSBF drops 3 times in each occasion, hence performing the best among the algorithms in terms of FNR.

We present the FPR and FNR of 695M records with 90% distinct element in Table 6. The memory size again varies from 64MB to 512MB. For various memory size, the algorithms exhibits similar trends as before with all the proposed variations providing large improvements in terms of FNR over SBF while keeping

Dataset:695M , Distinct:60%						
Space		SBF	RSBF	BSBF	BSBFSD	RLBSBF
64 MB	% FPR	6.7672	12.2408	16.0723	22.4491	22.7214
	% FNR	70.832	52.8855	39.9809	19.9394	20.2164
128 MB	% FPR	4.3075	6.7265	8.0154	9.7951	10.4734
	% FNR	57.9713	39.2682	28.6708	14.7555	9.8051
256 MB	% FPR	2.0861	2.7654	3.0816	3.4335	3.6588
	% FNR	42.0788	25.7648	17.7509	9.0574	3.4843
512 MB	% FPR	0.8151	0.9189	0.9779	1.0331	1.0775
	% FNR	24.2902	15.2035	9.8819	4.9892	1.0236

Table 5: Synthetic Dataset of 695M elements (60% distinct)

the FPR at comparable level. For example at the cost of 1.5x times worse FPR (from 1.6% to 2.2%), RLBSBF attains an improvement of 13x time in FNR over SBF.

Dataset:695M , Distinct:90%						
Space		SBF	RSBF	BSBF	BSBFSD	RLBSBF
64 MB	% FPR	8.5028	16.3502	20.6486	31.104	30.1024
	% FNR	73.134	55.0285	45.7102	24.1625	26.8786
128 MB	% FPR	6.3252	10.6716	11.9222	15.7484	16.6894
	% FNR	61.1759	41.9224	36.5123	20.3118	16.4363
256 MB	% FPR	3.5908	5.0678	5.3168	6.3016	6.8635
	% FNR	45.4987	28.179	25.3195	13.8078	7.0784
512 MB	% FPR	1.6058	1.8497	1.8911	2.0783	2.2248
	% FNR	26.6313	16.9302	15.3766	8.1061	2.3059

Table 6: Synthetic Dataset of 695M elements (90% distinct)

Table 7 presents the FPR and FNR with 1B records with 15% distinct element in the stream. The memory size varies from 64MB to 512MB for the underlying Bloom Filter based data structures. Here we observe that SBF performs very poorly in terms of FNR for low memory size. For 64MB memory, RLBSBF keeps the FPR and FNR to very low limits and achieves almost 30x improvement

over SBF in FNR while losing only 2.5x in FPR. As the memory size increases, FPR for all our algorithms becomes stable at a very low thresholds (0.1% – 0.2%) similar to that of SBF. But the gain in FNR performance is enormous. While SBF has a FNR of 17.1336% for 512MB memory, RSBF attains 10.2015% FNR, nearly a 2x improvement. BSBF, BSBFSD, RLBSBF observes an improvement of 17x times, 27x times and 317x times respectively.

Dataset:1B , Distinct:15%						
Space		SBF	RSBF	BSBF	BSBFSD	RLBSBF
64 MB	% FPR	2.9156	4.2891	5.5775	6.3441	6.6755
	% FNR	60.8981	43.1705	13.7096	4.6175	2.5795
128 MB	% FPR	1.2608	1.6079	1.9023	2.0181	2.0930
	% FNR	45.9566	29.5540	5.9096	2.4357	0.7400
256 MB	% FPR	0.4413	0.5059	0.5572	0.5727	0.5849
	% FNR	30.8269	18.1142	2.6956	1.2296	0.2026
512 MB	% FPR	0.1341	0.1431	0.1506	0.1526	0.1543
	% FNR	17.1336	10.2015	1.2846	0.6139	0.0535

Table 7: Synthetic Dataset of 1B elements (15% distinct)

Table 8 presents the FPR and FNR with 1B records with 60% distinct element in the stream. The memory size varies, as before, from 64MB to 512MB for the underlying Bloom Filter based data structures. Here also we observe FNR for SBF degrades to an unacceptable limit for 64MB or 128MB memory size. For 64MB memory, RLBSBF keeps the FPR and FNR to reasonable limits. But as the memory size increases, FPR for all our algorithms becomes stable at a very low point (1% – 2%) similar to that of SBF. But FNR performance is improved drastically for all our algorithms. While SBF has a FNR of 30.2739% for 512MB memory, RSBF attains an FNR of 20.2770%, nearly a 1.5 times improvement. BSBF, BSBFSD, RLBSBF observes an improvement of 2.5 times, 5 times and 30 times respectively. This improvements come at the cost of 2 time more FPR in the worst case (RLBSBF).

Table 9 presents the FPR and FNR with 1B records with 90% distinct element in the stream. The memory size varies, as before, from 64MB to 512MB. Here also we observe that at low memory only RLBSBF has both FPR and FNR at around 50%. All other algorithm exhibits either very poor FPR or very poor FNR. However as the memory size increases, FPR for all our algorithms becomes

Dataset:1B , Distinct:60%						
Space		SBF	RSBF	BSBF	BSBFSD	RLBSBF
64 MB	% FPR	7.8507	15.0561	20.7283	31.3858	30.044
	% FNR	75.7271	58.6734	44.9042	21.0835	25.7526
128 MB	% FPR	5.6378	9.5676	11.9472	15.5928	16.3554
	% FNR	65.2456	46.6885	34.7514	17.7438	15.0161
256 MB	% FPR	3.1629	4.5557	5.2369	6.0937	6.5378
	% FNR	50.5726	32.6139	23.2001	11.9067	6.2079
512 MB	% FPR	1.4504	1.6747	1.82011	1.9692	2.0788
	% FNR	30.2739	20.2770	13.5658	6.8855	1.9897

Table 8: Synthetic Dataset of 1B elements (60% distinct)

stable at 2% – 4% similar to that of SBF. While SBF attains a FPR of 2.72%, RSBF, BSBF, BSBFSD and RLBSBF also attains a FPR that is comparable with that of SBF. But the gain in FNR performance is huge. While SBF has a FNR of 32.8335% for 512MB memory, RSBF has 22% FNR, nearly a 1.5 times improvement. BSBF, BSBFSD, RLBSBF observes an improvement of 1.5 times, 3 times and 8 times respectively.

Dataset:1B , Distinct:90%						
Space		SBF	RSBF	BSBF	BSBFSD	RLBSBF
64 MB	% FPR	9.2617	18.6342	25.0083	40.5501	36.7959
	% FNR	77.5273	60.3738	49.2073	23.8819	31.0134
128 MB	% FPR	7.5909	13.8357	16.4207	23.192	23.5977
	% FNR	68.0553	49.2204	41.7178	22.89	22.2474
256 MB	% FPR	5.0231	7.7876	8.3846	10.4808	11.3415
	% FNR	54.0293	35.2653	31.2527	17.2991	11.5338
512 MB	% FPR	2.7427	3.2263	3.33317	3.7953	4.1133
	% FNR	32.8335	22.3998	20.2695	10.8729	4.2852

Table 9: Synthetic Dataset of 1B elements (90% distinct)

Hence, we observe that for both synthetic and real datasets of upto 1 billion records and varying percentage of distinct elements, the proposed algorithms in

this work clearly outperforms SBF in terms of FNR, attaining an improvement of more than 300x times in certain cases. Also, for reasonable amount of memory FPR performance of all the algorithms turn out to be similar. Coupled with enhanced convergence rates compared to that of SBF, we present novel and efficient algorithms for the de-duplication problem.

## 7. Conclusions and Future Work

Real-time de-duplication or data redundancy removal for streaming datasets poses a challenging problem. In this work we have presented novel Bloom Filter based algorithms to tackle the problem efficiently. Using a novel combination of reservoir sampling and Bloom Filters we have proposed RSBF to obtain enhanced FNR and faster convergence to stability at comparable FPR with that of SBF. We further proposed BSBF encompassing a biased sampling method with Bloom Filters to obtain better FNR for varied applications requiring very low FNR tolerance. Variations of BSBF have also been presented in this work with different deletion designs to counter the effects to multiple element deletion in Bloom Filters. Finally a randomized load balanced algorithms has also been presented to provide a balanced performance on both the FPR and FNR fronts. These features make the proposed algorithms extremely efficient and applicable to real life scenarios.

Using detailed theoretical results, we have proven the enhanced performance of the proposed algorithms in terms of FNR and convergence rates (stability). We demonstrate real-time in-memory DRR using both real and synthetically generated datasets of upto 1 billion records. We show FNR improvement over a vast range from 2x to 300x over existing results. To the best of our knowledge this work achieves the best FNR and convergence rates known with the same memory requirements as that of the competing algorithms. In future, we hope to study the effects of other biasing and sampling functions to further decrease the FNR. Investigations over the use of other structures and parallelizing the proposed algorithms may in turn lead to further enhancement and advancements in the field of parallel data redundancy removal research.

## References

- [1] V. K. Garg, A. Narang, S. Bhattacharjee, Real-time memory efficient data redundancy removal algorithm, in: CIKM, 2010, pp. 1259–1268.
- [2] A. Heydon, M. Najork, Mercator: A scalable, extensive web crawler, in: World Wide Web, Vol. 2, 1999, pp. 219–229.

- [3] A. Metwally, D. Agrawal, A. E. Abbadi, Duplicate detection in click streams, in: WWW, 2005, pp. 12–21.
- [4] H. Garcia-Molina, J. D. Ullman, W. J., Database System Implementation, Prentice Hall, 1999.
- [5] M. Bilenko, R. J. Mooney, Adaptive duplicate detection using learnable string similarity measures., in: Proc. SIGKDD, 2003, pp. 39–48.
- [6] M. Weis, F. Naumann, Dogmatrix tracks down duplicates in xml., in: Proc. ACM SIGMOD, 2005, pp. 431–442.
- [7] N. Alon, Y. Matias, M. Szegedy, The space complexity of approximating the frequency moments, in: STOC, 1996, pp. 20–29.
- [8] P. Gupta, N. McKeown, Packet classification on multiple fields, in: SIGCOMM, 1999, pp. 147–160.
- [9] J. Gehrke, F. Korn, J. Srivastava, On computing correlated aggregates over continual data streams, in: SIGMOD, 2001, pp. 13–24.
- [10] M. Reiter, V. Anupam, A. Mayer, Detecting hit-shaving in click-through payment schemes, in: USENIX, 1998, pp. 155–166.
- [11] A. Chowdhury, O. Frieder, D. Grossman, M. McCabe, Collection statistics for fast duplicate document detection, ACM Trans. on Information Systems 20 (2) (2002) 171–191.
- [12] J. Conrad, X. Guo, C. Schriber, Online duplicate document detection: Signature reliability in a dynamic retrieval environment, in: CIKM, 2003, pp. 443–452.
- [13] D. Lee, J. Hull, Duplicate detection in symbolically compressed documents, in: ICDAR, 1999, pp. 305–308.
- [14] A. Adya, W. J. Bolosky, M. Castro, G. Cermak, R. Chaiken, J. R. Douceur, J. Howell, R. J. Lorch, M. Theimer, R. Wattenhofer, Farsite: Federated, available, and reliable storage for an incompletely trusted environment., in: OSDI, 2002, pp. 1–14.
- [15] F. Douglass, J. Lavoie, J. M. Tracey, P. Kulkarni, P. Kulkarni, Redundancy elimination within large collections of files., in: USENIX, 2004, pp. 59–72.

- [16] N. Tolia, M. Kozuch, M. Satyanarayanan, B. Karp, T. C. Bressoud, A. Perrig, Opportunistic use of content addressable storage for distributed file systems, in: USENIX, 2003, pp. 127–140.
- [17] S. Quinlan, S. Dorward, Venti: A new approach to archival storage, in: FAST, 2002, pp. 89–101.
- [18] N. Jain, M. Dahlin, R. Tewari, Taper: Tiered approach for eliminating redundancy in replica synchronization, in: FAST, 2005, pp. 281–294.
- [19] B. H. Bloom, Space/time trade-offs in hash coding with allowable errors, *Commun. ACM* 13 (7) (1970) 422–426.
- [20] A. Z. Broder, M. Mitzenmacher, Network applications of bloom filters: A survey, *Internet Mathematics* 1 (4) (2003) 485–509.
- [21] L. Fan, P. Cao, J. Almeida, Z. Broder, Summary cache: a scalable wide area web cache sharing protocol, in: *IEEE/ACM Transaction on Networking*, 2000, pp. 281–293.
- [22] M. Mitzenmacher, Compressed bloom filters, in: *IEEE/ACM Transaction on Networking*, 2002, pp. 604–612.
- [23] A. Kumar, J. Xu, J. Wang, O. Spatschek, L. Li, Space-code bloom filter for efficient per-flow traffic measurement, in: *IEEE INFOCOM*, 2004, pp. 1762–1773.
- [24] C. Saar, M. Yossi, Spectral bloom filters, in: *ACM SIGMOD*, 2003, pp. 241–252.
- [25] H. Shen, Y. Zhang, Improved approximate detection of duplicates for data streams over sliding windows, *J. of Computer Science and Technology* 23 (6).
- [26] W. Feng, D. Kandlur, D. Sahu, K. Shin, Stochastic fair blue: A queue management algorithm for enforcing fairness, in: *IEEE INFOCOM*, 2001, pp. 1520–1529.
- [27] F. Baboescu, G. Varghese, Scalable packet classification, in: *ACM SIGCOMM*, 2001, pp. 199–210.



- [28] S. Dharmapurikar, P. Krishnamurthy, D. Taylor, Longest prefix matching using bloom filters, in: ACM SIGCOMM, 2003, pp. 201–212.
- [29] Y. Hua, B. Xiao, A multi-attribute data structure with parallel bloom filters for network services, in: International Conference on High Performance Computing, 2006, pp. 277–288.
- [30] M. Little, N. Speirs, S. Shrivastava, Using bloom filters to speed-up name lookup in distributed systems, *The Computer Journal* (Oxford University Press) 45 (6) (2002) 645 – 652.
- [31] F. Deng, D. Rafiei, Approximately detecting duplicates for streaming data using stable bloom filters, in: SIGMOD, 2006, pp. 25–36.
- [32] P. Flajolet, G. N. Martin, Probabilistic counting algorithms for database applications., *Comput. Syst. Science* 31 (2) (1985) 182–209.
- [33] B. Babcock, M. Datar, R. Motwani, Sampling from moving window over streaming data, in: SODA, 2002, pp. 633–634.
- [34] P. Gibbons, Y. Mattias, New sampling-based summary statistics for improving approximate query answers, in: ACM SIGMOD, 1998, pp. 331–342.
- [35] P. Gibbons, Distinct sampling for highly accurate answers to distinct value queries and event reports, in: VLDB, 2001, pp. 541–550.
- [36] C. Aggarwal, P. Yu, *Data Streams: Models and Algorithms*, Springer, 2007.
- [37] J. S. Vitter, Random sampling with a reservoir, *ACM Trans. on Mathematical Software* 11 (1) (1985) 37–57.
- [38] C. C. Aggarwal, On biased reservoir sampling in the presence of stream evolution, in: VLDB, 2006, pp. 607–618.
- [39] S. Dutta, S. Bhattacharjee, A. Narang, Towards ”intelligent compression” in streams: A biased reservoir sampling based bloom filter approach, in: EDBT, 2012, pp. 228–238.